Preprint
UCRL-JC-141943

# ROSETTA: The Compile-Time Recognition of Object-Oriented Library Abstractions and their use within user Applications

*D. Quinlan, B. Philip*

This article was submitted to
Joint Association for Computing Machinery Java Grande-International Scientific Computing in Object-Oriented Parallel Environments Conference, Palo Alto, CA, June 2-4, 2001

**January 8, 2001**

# ROSETTA: The Compile-Time Recognition Of Object-Oriented Library Abstractions And Their Use Within User Applications *

Dan Quinlan and Bobby Philip
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
dquinlan,bobbyp@llnl.gov

## ABSTRACT

Libraries arise naturally from the increasing complexity of developing scientific applications, the optimization of libraries is just one type of high-performance optimization. Many complex applications areas can today be addessed by domain-specific object-oriented frameworks. Such object-oriented frameworks provide an effective compliment to an object-oriented language and efficitvely permit the design of what ammount to essentially domain-specific languages. The optimization of such a domain-specific library/language combination however is particularly complicated due to the inability of the compiler to optimize the use of the libraries abstractions.

The recognition of the use of object-oriented abstractions within user applications is a particularly difficult but important step in the optimization of how objects are used within expressions and statements.

Such recognition entails more than just complex pattern matching. The approach presented within this paper uses specially built grammars to parse the C++ representation. The C++ representation is itself obtained using a modified version of the SAGE II C/C++ source code restructuing tool which is inturn based upon the Edison Design Group (EDG) C++ front-end. ROSETTA is a tool which automatically builds grammars and parsers from class definitions, associated parsers parse abstract syntax trees (ASTs) of lower level grammars into ASTs of higher level grammars. The lowest level grammar is that associated with the full C++ language itself, higher level grammars specialize the grammars specific to user defined objects. The grammars form a hierarchy and permit a high-degree of specialization in the recognition of complex use of user defined abstractions.

*This work is funded (in part?) by the Department of Energy's Division of Mathematical, Information, and Computational Sciences under contract number ???.

## 1. INTRODUCTION

The optimization of object-oriented libraries and particularly the applications that use them has been a longstanding roadblock to the devleopment of object-oriented scientific applications. The desire to encapsulate the increasing complexity of parallel scientific computations is a natural driving force in the continued development of object-oriented libraries and frameworks for scientific computing. But the inability of the C++ compiler to optimize the use of abstractions from such libraries and frameworks forces the use of awkward lowlevel interfaces or poor performance from high-level interfaces. A standing problem in the development of compile-time optimization of user defined object-oriented abstractions is the inability of the C++ compiler to recognize and optimize their use with applications. That they are unoptimized by the C++ compiler is essentially because they are *user defined.*

The preprocessor approach to optimizations is dificult because of the complexity of recognizing the use of user defined abstractions within an application. ROSETTA is a tool for automatically building anotated C++ grammars and parsers that can recognize complex use of user defined abstractions at compile-time. Coupled with a separate tool for introducing source-to-source transformations, ROSE, entire source-to-source compilers can be constructed that operate aheadof the C++ compiler and optimize the performance of user defined object-oriented libraries. This paper details the use and construction of the recognition phase of this optimization approach.

A *Meta-program* level is used to define the preprocessor, this level is a simple C++ application code. The *Meta-program* defines the manipulation of grammars using the **ROSETTA** library. The output of the *Meta-program,* when it is executed, is source code (written to files). The source code is compiled, with the ROSE infrastructure, to form a preprocessor specific to a given framework. The *Meta-program* can generate a lot of source code, typically 200,000 lines, but it can be compiled in under a minute and once built into a preprocessor need not be recompiled by the user.

## 2. ROSETTA

**ROSETTA** is a tool we developed for the manipulation of grammars. It permits a C++ *Meta-program* to be defined which, when executed, builds tools like Sage II. It is not a novel part of this work to have defined a mechanism to generate the Sage II source, modified or not. It **is** a novel part of this research work that higher-level grammars can be automatically generated in addition to the Sage II source. This important feature is the mechanism by which critical parts of the preprocessor are customized for a framework's abstractions; and automatically generated.

**ROSETTA** represents a class library of terminals and non-terminals used to define a grammar. It is relatively trivial to define the C++ grammar in terms of terminals and nonterminals and associate with the terminals and nonterminals application code. We consider an implementation of the grammar to be a library of classes representing the different language elements defined by a grammar (statements, expressions, types, etc.). We use the Sage II library as a basis for our C++ grammar, but other libraries that implement grammars and form the basis of different sorts of compiler tools exist[?, ?].

### 2.1 Generation of the C++ Grammar's Implementation

Figure 1 shows the use of ROSETTA in the *Meta-program* for the construction of the modified version of Sage II.

**figure***

Here the example program builds the implementation of the C++ grammar (mostly represented as a copy of the Sage II source code with modifications). The output of this application is about 70,000 lines of source code. With the output files compiled into a preprocessor and linked with the ROSE infrastructure, the final preprocessor parses C++ applications and unparses them to generate C++ (identical to the input code in format as well as syntacticly). Such a preprocessor is of little use for our purposes but forms a trivial example of a preprocessor built using ROSE.

## 3. SPECIFICATION OF THE BASE LEVEL (C++) GRAMMAR

This section explains the construction of the C++ grammar. This step is particularly simple, the default constructor for the Grammar class builds the C++ grammar. The code which is produced is essentially a modified form of the Sage II source code. The C++ code produced in this step can be compiled with the rest of the ROSE infrastructure to produce a preprocessor that parses a C++ application into the C+ grammar as defined by Sage II. This step represents steps 1 and 2 above. Numerous features at this level are available:

1. Traversal of the AST to do program analysis
2. Editing of the AST to introduce transformation

The remaining problem with this level of representation of the users application code's AST is that:

1. it is *very* large
2. a framework's high-level abstractions are hidden in the C++ syntax
3. the interactions between a framework's abstractions are hidden particularly deep in the C++ syntax.

In principle it should not be difficult to recognize arbitrary high-level abstractions and their interactions, but our efforts demonstrated that it is full of practical limitations. Hence the alternative approach we have implemented and present in this paper.

### 3.1 Generation of a High Level Grammar's Implementation

This section explains the system of constraints used to define higher level grammars (higher level and more specific than the C++ grammar). The principle is to include and exclude terminals in an existing grammar (the Base grammar for our purposes is the C++ grammar). Terminals are added or removed as desired to define modifications of the C++ grammar. As an example, additional terminals can be added to define additional types represented by a class defined within an object-oriented framework. New terminals are added through the specification of an existing C++ terminal *plus* constraints. The form of the constraints can be varied (and are expressed using C++ code).

As an example, the specification of a class name could be used to define a new terminal in a new grammar specific to a class name associated with a framework's abstraction (assuming the abstraction is an object). The result is a grammar for which the framework's abstraction is recognized as an implicit type within the higher-level grammar. The use of the framework's abstraction within expressions can be recognized through the addition of expression terminals to the higher-level grammar. Since all elements of the higher-level grammar are built from terminals of the C++ grammar with an additional constraint no modifications to the C++ language are possible. This is a strength of this mechanism since we want to recognize a framework's abstractions and not formally extend the C++ language.

To further customize the high-level grammar to a particular framework's abstractions, the addition of a new type terminal drives the automated introduction of all possible expression terminals with the constraint that they are between objects of the new added type. The classes represented by the new types are further interrogated to define all possible expressions (member functions of the framework's abstraction) represented by the new type. Similarly statement terminals are added to represent statements containing expressions in the new type. Since the addition of new types adds to the number of terminals (and non terminals) in a grammar, the size of the grammar's implementation nearly doubles. Since this step is fully automated, the amount of additional code generated is not important. Within this approach, through the design of the higher level grammars, we permit user defined types and their expressions and statements to be treated as implicit keywords within an user's application.

```
// include definitions of grammars, terminals, and non-terminals
// (objects within ROSETTA)
#include "grammar.h"

int main()
   {
  // Build the C++ grammar (generate Sage II source)
     Grammar sageGrammar;

  // Build the header files and source files
  // representing the grammar's implementation
     sageGrammar.buildCode();
   }
```

**Figure 1: Example Meta-Program for the generation of C++ Grammar (esentially a modified form of SAGE II).**

## 3.2 Extending the Base Grammar

Terminals and non-terminals can be easily added to the existing grammar by the specification of new terminals.

- A. The inclusion of a new statement type for example adds an X statement to the existing grammar.

  Figure 2 shows how to add a new statement to a grammar.

  **figure***

- B. The addition of a new expression to an existing grammar is more interesting.

  Figure 3 shows how to add a new statement to a grammar.

  **figure***

  An consequence is that the addition of a new expression element forces the creation of any of any possible statement (since expressions can be combined into statements in so many ways). So a set of all possible statement terminals is defined whenever an expression terminal is defined.

- C. The addition of a new type occurs through a well defined mechanism (the addition of a new class type). Consequently, with the addition of a new type and since expressions could be defined with objects of the new type, a whole set of expressions are added to the new grammar. Just as with the addition of a new expression (above) the addition of the set of expressions forces the addition of all possible statements that can be defined from the new expressions.

  Figure 4 shows how to build a new higher level grammar specific to a given user defined array class.

  **figure***

  Figure 5 shows a shorter form is possible if we are just interested in adding a type is a specialized member function (this is an interface issue only).

  **figure***

## 3.3 Restrictions of High Level Grammars

At this point the creation of higher level grammars is largely dubious since we have only defined a mechanism to recognize the occurance of a user defined type within a program tree (AST). A similar result could be obtained (perhaps not as elegantly) through a traversal of the program tree. The point is to go much farther in the definition of the higher level grammar and to both expand and restrict the added elements defining the higher level grammar.

The restriction of a user defined type's use is important in how it is applied in expressions to define optimizible statements. For new user defined types representing element wise semantics, Statements containing expressions with user defined functions must disqualify optimization (though a more complex transformation can be developed). Figure 6 shows a disqualifying case.

**figure***

demonstrates an example of how the higher level grammar must remove this terminal from it's representation within the expanded part of the higher level grammar. Thus without it's representation within the grammar such a defined function would not permit the definition of an expression as a special (e.g. "X" expression) within the program tree.

Figure 7 shows an example of restriction (removal of elements from the grammar).

**figure***

## 3.4 Consolidation of types within expanded grammar

Characteristics of several terminals (within the extended grammar) can be consolitated within a single terminal. For example the different variations of loop constructs (for statement, while statement, do-while statement) can be consolidated within a single terminal within the extended grammar.

Figure 8 shows such a consolidation of terminals into a single terminal.

**figure***

## 3.5 Specification of Predefined Semantics

Although a classification of semantics is not clear at this point, we can greatly simplify the specification of numerous specialized semantics associated with algebras defined

```
   Terminal & loopStatement = ForStatement | WhileStatement | DoWhileStatement;
   X_Grammar.addTerminal(loopStatement);
```

Figure 2: Example showing the construction of terminal for a loopStatement.

```
   Terminal & functionExpression = sum | product | reciprocal;
   X_Grammar.addTerminal(functionExpression);
```

Figure 3: Example showing the construction of terminal for an expression.

```
   Grammar sageGrammar ("Cxx_Grammar","Sg","ROSE_BaseGrammar");

// Build the header files and source files representing the grammar's implementation
   sageGrammar.buildCode();

   Grammar X_Grammar ("X_Grammar","XG_","ROSE_BaseGrammar",&sageGrammar);

// Build a new terminal as a copy of an existing terminal (this leverages the
// existing terminals implementation and can be sublemented with constraints).
// Copy a terminal and give it a new name (with a new tag name).
// The copy is then a child of the copied terminal, parsing the parent triggers the
// parsing of the children (constraints are tested and a child is built if a constraint
// test passes, else the parent is built).
// In the tree hierarchy the new terminal is DERIVED from the parent (thus the doubleArrayType
// is derived from the ClassType.  This makes sense because the doubleArrayType is a
// "specialization" of the X_ClassType terminal).
   Terminal & doubleArrayType = X_Grammar.getTerminal("ClassType").copy("doubleArrayType","DOUBLE_ARRAY_TYPE_TAG")

// Build a constraint (using the SAGE interface) and add it to the new terminal.
   char* constraintString = "isSgClassDeclaration() && isSgClassDeclaration()->getName() == \"doubleArray\"";
   doubleArrayType.addConstraint("declaration",constraintString);

// Adding a terminal to the grammar will automatically place the
// terminal in the correct location within the tree hierarchy
   X_Grammar.addNewTerminal(doubleArrayType);

// Build the header files and source files representing the grammar's implementation
   X_Grammar.buildCode();
```

Figure 4: Example showing the construction of higher level grammar for a user defined array class.

```
   X_Grammar.addType("doubleArray");
```

Figure 5: Example showing the addition of a new type to a grammar.

```
 A = B + foo(); // example showing the use of a user defined
                // function foo() which can't be optimized directly.
```

Figure 6: Example showing case that would not be classified as an array statement.

```
   X_Grammar.removeType("doubleArray");
```

Figure 7: Example showing the removal of a type from the grammar.

```
   Terminal & forStatement     = X_Grammar.getTerminal("X_ForStatement");
   Terminal & whileStatement   = X_Grammar.getTerminal("X_WhileStatement");
   Terminal & doWhileStatement = X_Grammar.getTerminal("X_DoWhileStatement");
   Terminal & loopStatement    = forStatement | whileStatement | doWhileStatement;
   X_Grammar.addNewTerminal(loopStatement);
```

Figure 8: Example showing the consolidation of terminals into a single terminal.

on objects (e.g. arrays, particles, etc which are common abstractions within scientific computing).

#### *3.5.1 hasCollectionElementSemantics*
Collection Semantics imply that operations defined on the collection apply element-wise to each of the elements in the collection. Array objects typically contain such semantics. The specification of this predefined semantics simplifies the definition of transformations on objects representing collections.

#### *3.5.2 hasArraySemantics*
Array semantics imply that operations on the Rhs of an assignment are completed before any assignment to the Lhs.

#### *3.5.3 hasAsynchronousExecutionSemantics*
Asynchronous Execution Semantics imply that operations between different operands are independent. Thus the dependence can be fully resolved through an analysis of the dependence graph and alias analysis. This is common within many array classes for example.

#### *3.5.4 Examples*
The following example show how such predefined semantics are specified for a grammar build around the definition of such a type. Figure 9 shows an example.

**figure***

### 3.6 Examples
Adding a new type and Removing user defined functions returning new type

## 4. IMPLEMENTATION
The implementation of ROSETTA builds upon SAGE II [?], which is built upon the Edison Design Group (EDG) C++ front-end. Our work has been greatly simplified by access to these two tools. ROSETTA uses a modified form of the SAGE II which we have developed. The purpose was to

- Permit the automate generation of what is essentialy a modified version of SAGE II
- Maintain the SAGE II source code (so that we can fix minor bugs and make additions (templates, and support for new C++ features as supported by EDG))
- Introduce the use of STL (as an outside library) into the design of SAGE II
- Remove as many asymetries from the implemention of SAGE II so that the generation of the code could be simplified.
- Modify the SAGE II source to permit it to be used as a basis for all higher level grammars. This required naming the classes so that multiple grammars could coexist (to build hierarchies of grammars) in the same source-to-source compiler.

While using SAGE II as a basis for the grammars that ROSETTA generates. ROSETTA the significant capability to define grammers at the level of BNF notation. C++ classes are used to represent terminals and non-terminals within the grammar.

Operators (overloaded operators defined for the terminal and non-terminal classes) are used to define production rules.

## 5. RESULTS
We have build high level grammars and used them to recognize and classify the use of user defined abstractions with numerous applications. The appraoch is particularly simple since the grammars can be built automatically from the library header files where the classes are defined. Some additional information is required if numerous default definitions are to be overridden. It is not possible within this paper to present the ASTs for the higher level grammers since graphs as complex as these are difficult to visualize and we currently lack mechanisms for there presentation except for debugging purposes.

## 6. CONCLUSIONS
The use of object-oriented frameworks can often require compile-time optimization if the abstractions are not suffiently coarse grain and the context of the abstractions use is important. This is the case for numerous sorts of abstractions for which the statements that use them consist of multiple expressions. Alternatively, blocks of statements may benifit from optimizations where there context relative to one another can be optimized. Examples could be array class, matrix classes, particle classes, finite-element classes, etc. Since a library can not readily see the context of how it's elements are juxaposed, only a compile-time tool can readily disern the use of object-oriented abstractions within a user's application. With the abstract syntax tree exposed, clearly a relatively simple pattern matching approach could be used to identify the objects within an applications, but this is not enought to be useful. To recognize where transformations can be introduced it is required that the use of the object-oriented abstractions be identified and classified (into specific sorts of expressions, statements, types, symbols, etc.). With this level of detail the AST is greatly simplified and can be traversed with the intend of optimization, syntax checking, etc. This level of recognittion is of far greater sophistication than searching for funtion names as might be the limits of the requirements of C or FORTRAN library optimizations. The sophisticated level of user defined abstraction recognition presented within this paper is likely just as applicable to JAVA, though the extension to other languages has not be our focus. The target of its use so far has been within the ROSE source-to-source optimizing compiler infrastructure.

```
// Here we endow the "X" class with predefined semantics
// to simplify the specification of the transformations
   X.hasCollectionElementSemantics(TRUE);
   X.hasArraySemantics(TRUE);
   X.hasAsynchronousExecutionSemantics(TRUE);
```

**Figure 9: Example showing specification of semantics in the grammar.**